

## 14 Factoring

One of the more interesting problems with which to study parallelism is the problem of factoring.

**Problem:** Given an integer  $N$ , factor that integer into a product of prime numbers.

First, let's consider the nature of the problem. We know that  $N$  has at least one prime factor smaller than the square root of  $N$ . (Proof: If  $N$  is not prime, it has at least two prime factors. If both are larger than the square of  $N$ , then their product is larger than  $N$ . This is a contradiction.)

Second, we're going to benchmark the nature of the factoring problem using  $N \approx 2^{512}$  for analysis purposes. A number of this size can be factored (look up H.J.J. te Riele at the CWI in Amsterdam to see how he did it), but it takes enormous effort (i.e., some months of computing on at least some dozens of machines).

Also, we have to consider whether there is anything we know about  $N$  beforehand. In the "old days" when factoring was just something done by number theorists for enjoyment (and the occasional published paper), we might be given integers  $N$  about which nothing would be known. Nowadays, given the connection between factoring and public key cryptography, we might know something. If in fact an integer to be factored comes from a public key problem, then it is probable that  $N$  has been constructed so as to be difficult to factor. And in general, the hardest numbers to be factored are those that are the product of two primes of roughly equal size.

We will cover two methods of factoring that are not guaranteed to work, but that are relatively cheap to run. Even if one is handed a number that

one suspects is a cryptographic number, it's worth trying both these methods first. The guaranteed-to-work method takes, as mentioned above, some weeks of compute time at least on some dozens of machines. With that kind of effort staring one in the face, running one of the other methods overnight in hopes of getting lucky is definitely worth the effort.

### 14.1 Divide by primes

One obvious approach is to divide  $N$  by the primes less than or equal to the square root of  $N$ . This would have the advantage of being highly parallel. We could think of using a massively parallel computer that had nothing but dividers in it. Pass out a prime to each processor and do the division. If any prime  $p$  divides evenly, then we have a factor of  $N$  and the cofactor  $N/p$  is yet another number to be factored, but a smaller number, so the problem has been reduced to a simpler one.

The problem with this method is that it just isn't efficient enough. Let  $\pi(x)$  be the number of primes less than or equal to  $x$ . The Prime Number Theorem (proved by Hadamard and de la Vallee Poussin in 1896) says  $\pi(x) \approx \frac{x}{\log x}$ . Using our benchmark  $N$  of about 512 bits, there's not guaranteed to be a factor until we search up through primes of about 256 bits. Now,

$$\frac{2^{256}}{\log 2^{256}} - \frac{2^{255}}{\log 2^{255}} \approx .72 \times 2^{248}.$$

That is, there are about  $2^{248}$  primes of size 256 bits. Trial division just requires dividing by way too many primes to be at all feasible, even if the divisions could be done in parallel.

## 14.2 Group Theoretic Methods

We will start by looking at a couple of group theoretic methods. Recall that the integers modulo a prime  $p$  form a multiplicative cyclic group of  $p - 1$  elements. That is, they form a single multiplicative cycle, and the cycle can be generated by any primitive root. We have Fermat's Little Theorem (which is also Lagrange's theorem in finite groups): Given an integer  $a$ , then if  $p$  is a prime, we have  $a^{p-1} = 1$  modulo  $p$ . Obviously, then, if  $p - 1$  divides some integer  $M$  evenly, then  $a^M = 1$  modulo  $p$  simply because  $a^M = (a^{p-1})^{M/(p-1)} = 1^{M/(p-1)} = 1$ .

### 14.2.1 Pollard's $p - 1$ Factoring Method

To factor  $N$ , compute  $M$  as the product of all small primes raised to all reasonable high powers. (For example, if we take all primes less than one million raised to the highest power that still fits in a 32 bit word, and multiply all of these together to get  $M$ , then  $M$  is a number of about 3 million bits.) Then pick any random integer  $a$  modulo  $N$ , and compute  $a^M$  modulo  $N$ .

If  $N$  has a factor  $p$  for which  $p - 1$  divides  $M$ , then  $a^M$  modulo  $N$  will be 1 modulo  $p$ . If we subtract 1, and take a greatest common divisor with  $N$ , then we should get  $p$  as a factor of  $N$ .

Comment: Although we can think for convenience about the single cycle of powers generated sequentially, as in the following example, remember that it is possible to exponentiate by recursive doubling. If the exponent is  $M$ , then one needs to write  $M$  in binary as a number of  $b = \lg M$  bits. To compute the exponentiated value, one needs to square  $b$  times and to multiply in by a running product for each 1-bit in the exponent (roughly  $b/2$  times).

Thus we only need roughly  $\frac{3}{2} \lg M$  multiplications (if we count squarings as multiplications).

### 14.2.2 Example

Let  $N = 437 = 19 \times 23$ . If we take powers of 2, we get the following.

exponent	power modulo $N$	power mod 19
1	2	1
2	4	4
3	8	8
4	16	16
5	32	13
6	64	7
7	128	14
8	256	9
9	75	18
10	150	17
11	300	15
12	163	11
13	326	3
14	215	6
15	430	12
16	423	5
17	409	10
18	381	1

Now, what are the characteristics of this algorithm? First, the compu-

tations performed are simple, just being multiplications and exponentiation. Second, there's a way to get a factor of  $N$  from having computed the identity of the group. Finally, the order of the group is only as big as the prime  $p$  dividing  $N$ .

Comment: Note that Pollard's method only works if  $p - 1$  divides  $M$ . If all the prime factors of  $N$  are of the form  $p$  such that  $p - 1$  has a relatively large prime factor, then this method will not find a factor. We could, if we wanted to, examine analytically the likelihood that we could factor  $N$  with a particular exponent  $M$ . We won't do that. It will suffice for this course simply to point out that  $p - 1$  must be composed only of small factors.

In general, though, Pollard's method is a relatively nice way to extract smallish factors of  $N$ .

### 14.2.3 The Elliptic Curve Method

The  $p - 1$  method has the advantage of being simple, but it has the disadvantage that it doesn't always work. Further, it doesn't parallelize. In essence, it's a method that works only if one particular group, the group of linear residues modulo a prime factor  $p$  of  $N$ , is such that  $p - 1$  is a product only of small primes.

Let's consider a different method that has the advantage of allowing us to work in parallel in many different groups, only one of which has to have order equal to a product of small primes.

Let  $A$  and  $B$  be integers taken modulo a prime  $p$ . Consider the points

$(X_i, Y_i)$  that are the solutions to the equation

$$Y^2 = X^3 + AX + B$$

We will call this equation with its associated solutions an *elliptic curve*. It turns out that we can “add” these points in the following way.

#### 14.2.4 Obligatory picture

#### 14.2.5 Formulas

Given two points  $P_1 = (x_1, y_1)$  and  $P_2 = (x_2, y_2)$ , we compute the slope

$$m = \frac{3x_1^2 + A}{2y_1}$$

if  $x_1 = x_2$  or

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

otherwise. Then the “sum”  $P_3 = P_1 + P_2$  is  $P_3 = (x_3, y_3)$  where

$$x_3 = m^2 - x_1 - x_2$$

and

$$y_3 = m(x_1 - x_2) - y_1.$$

These are in fact just the formulas that correspond to the geometric picture.

**14.2.6 Example**

Let  $p = 11$  and let the curve be  $Y^2 = X^3 + 5X + 6$ . Then we can write the group of points on the curve as follows.

$$\begin{aligned}P & (1, 1) \\2P & (3, 2) \\3P & (10, 0) \\4P & (3, 9) \\5P & (1, 10)\end{aligned}$$

Now, the reason that we care about this is the following.

1. We choose a simple point  $P = (x_0, y_0)$  and simple parameters  $A$  and  $B$  so that the point  $P$  is on the “elliptic curve”  $E : Y^2 = X^3 + AX + B$ .
2. We don’t know the factors of  $N$ , but we can nonetheless work modulo  $N$  and compute multiples of  $P$  on the curve, just as we did with the Pollard  $p - 1$  method.
3. There’s a major theorem that says that the number of elements in the cycle of points modulo a prime  $p$  is about  $p \pm \sqrt{p}$ . So if  $N$  has a smallish prime factor  $p$ , then the length of the cycle on the elliptic curve is relatively small compared to the size of  $N$ .
4. We can compute  $M \cdot P$  for a very large composite  $M$  just as we computed  $a^M$  for Pollard  $p - 1$ , but recursive doubling.
5. If we work modulo  $p$ , then when we hit the identity of the group we get  $1/0$  modulo  $p$ . But if we work modulo  $N$  with  $p$  dividing  $N$ , we

don't get something that is zero mod  $N$  but rather something that is only zero mod  $p$ . Once again, if we take a gcd with  $N$  we'll pull off the factor of  $N$ .

So working, for example, modulo  $N = 187$  on the curve  $E : Y^2 = X^3 + 5X + 182$ , we would get

$$\begin{aligned} P & (1, 1) \\ 2P & (14, 2) \\ 3P & (120, 154) \end{aligned}$$

What happens next is that we try to keep on with the group operations and we find that we can't do it; we will get a "slope" of  $153/119$  and won't be able to rationalize the denominator because 119 and 187 aren't relatively prime.

From the point of view of parallel implementations, the big deal with the elliptic curve method is that we can use many parallel curves for many different parametric values  $A$  and  $B$ . This is naive parallelism, but it works. There are various ways in which one can determine, for a given  $N$ , the right number of curves to use in order to be assured that factors that can be found with this method will in fact be found with this method.

### 14.3 The Quadratic Sieve Method

The workhorse factoring method is the *Number Field Sieve*. The quadratic sieve is not really the same thing as the NFS, but it's close enough to illustrate the method.

The basic idea is that if we could write

$$x^2 - y^2 = N$$

then we could factor  $N$ , since we'd be able to factor algebraically

$$(x - y)(x + y) = N.$$

Now, it turns out that finding an *exact* solution as a difference of squares is very hard. But finding a *congruence*

$$x^2 - y^2 \equiv 0 \pmod{N}$$

is much easier, and we still get to factor algebraically to get

$$(x - y)(x + y) \equiv 0 \pmod{N}.$$

If we're very unlucky then we'll have  $x - y = 1$  and  $x + y = N$ . But unless this happens, we'll get one nontrivial factor of  $N$  in  $x - y$  and the other nontrivial factor of  $N$  in  $x + y$ .

We will illustrate the Quadratic Sieve by factoring 1147.

First, compute  $R = \sqrt{1147}$  and truncate this to the integer 33. Now, we will choose as a *factor base* the "small" primes, using as our definition of "small" the primes up through 19.

Now, run a loop and factor residues as follows:

	-1	2	3	11	13	17	19
$(33 - 6)^2 - 1147 = -418$	1	1	0	1	0	0	1
$(33 - 5)^2 - 1147 = -363$	1	0	1	2	0	0	0
$(33 - 4)^2 - 1147 = -306$	1	1	2	0	0	1	0
$(33 - 3)^2 - 1147 = -247$	1	0	0	0	1	0	1
$(33 + 2)^2 - 1147 = 78$	0	1	1	0	1	0	0
$(33 + 5)^2 - 1147 = 297$	0	0	3	1	0	0	0
$(33 + 6)^2 - 1147 = 374$	0	1	0	1	0	1	0

(I have only shown those lines that might be useful.)

Now, if we reduce this matrix (using Gaussian elimination, for example) to get a line in the matrix in which all the coefficients are zero mod 2, then it means that the product of the corresponding lines on the left hand column of the matrix factors, modulo  $N$ , as a product of primes with only even exponents. And something with only even exponents is a square.

For example, if we add the  $-6$ ,  $-3$ ,  $2$ , and  $5$  lines together, we get a line whose exponents on primes entries are as follows.

	-1	2	3	11	13	17	19
$(33 - 6)^2 - 1147 = -418$	1	1	0	1	0	0	1
$(33 - 3)^2 - 1147 = -247$	1	0	0	0	1	0	1
$(33 + 2)^2 - 1147 = 78$	0	1	1	0	1	0	0
$(33 + 5)^2 - 1147 = 297$	0	0	3	1	0	0	0
	2	2	4	2	2	0	2

What this means, then, is that

$$(33-6)^2(33-3)^2(33+2)^2(33+5)^2 \equiv (-1)^2(2)^2(3)^4(11)^2(13)^2(19)^2 \pmod{1147}$$

We rewrite this as

$$(27 \cdot 30 \cdot 35 \cdot 38 + 2 \cdot 3^2 \cdot 11 \cdot 13 \cdot 19)(27 \cdot 30 \cdot 35 \cdot 38 - 2 \cdot 3^2 \cdot 11 \cdot 13 \cdot 19) \equiv 0 \pmod{1147}$$

which becomes

$$(267 + 732)(267 - 732) \equiv (999)(-465) \equiv 0 \pmod{1147}$$

We compute the greatest common divisor of 999 and 1147 to get 37, and similarly the greatest common divisor of 465 and 1147 to get 31. Note that  $1147 = 31 \cdot 37$ .

### 14.3.1 Implementation Issues for the Quadratic Sieve

In what we have just described, there is what looks like a trial division step to factor the residues ( $-418, -363, -306$ , etc., in our example). This would be a problem, if in fact we had to do a trial division. We can avoid that in the following way.

Set up and initialize to zero an array indexed by the  $i$  values (as in  $R + i$ ). Now we loop through the primes in the factor base. For each prime  $p$ , compute a starting location for which  $p$  must divide  $(R + i)^2 - N$ . Then, for that value of  $i$  and at stride  $p$  up and down the array, add in the log of  $p$ . If, at the end of our loop through the primes in the factor base, we have in our array a value that is close to  $\log N$ , then we assume that we have factored the residue completely, and we keep that line in our matrix.

For example, if  $i$  is odd, then  $R+i$  and its square are odd, so  $(R+i)^2 - 1147$  is even. And if  $i$  is even, then  $(R+i)^2 - 1147$  is odd. So we can check off

the odd subscripts as having factors of 2 and the even subscripts as being prime to 2. Similarly, we can check that 3 divides  $(R + i)^2 - 1147$  for  $i = \dots, -8, -5, -2, 1, 4, 7, \dots$  as well as for  $i = \dots, -7, -4, -1, 2, 5, 8, \dots$

The upshot of this is that trial division is unnecessary.

### 14.3.2 The Multiple Polynomial Quadratic Sieve Method

Now, what we have described above is the vanilla quadratic sieve method. It works, but it's slower than its usual multiple polynomial version. In essence, we're using the polynomial

$$x^2 - N$$

to generate residues that we hope will be small enough to factor completely using only primes in the factor base. We can, with some analysis, determine the likelihood that with a given factor base we can succeed with an array of length  $2M$  for  $x = R - M, R - M + 1, \dots, R - 2, R - 1, R, R + 1, R + 2, \dots, R + M$ .

The multiple polynomial version has us choose a sequence of polynomials

$$Ax^2 + Bx + C$$

such that  $B^2 - 4AC = 4N$  so we can run multiple arrays at the same time (note that our original polynomial has  $A = 1, B = 0$ , and  $C = N$ ). With the right choice of coefficients  $A, B, C$ , we can force the residues to be "small" for multiple polynomials and for long sequences of  $x = R - M, R - M + 1, \dots, R - 2, R - 1, R, R + 1, R + 2, \dots, R + M$ .

## 14.4 An Analysis of Factoring Methods

It is worthwhile stepping back and thinking about the kinds of computations that are performed in these various algorithms and the kinds of parallelism that one can thus apply to the problem of factoring.

### 14.4.1 Pollard $p - 1$ and Elliptic Curve:

The fundamental operation here is a multiprecise multiplication. Whether one is computing  $a^M \pmod{N}$  in Pollard's  $p - 1$  method or one is computing  $M \cdot P$  for a point  $P$  on an elliptic curve, the basic "hard" step is the multiprecise multiply. To do this efficiently one needs a very fast processor for doing *integer* arithmetic, preferably one that handles add-with-carry or multiply-with-carry operations in one step.

As far as parallelism goes, there isn't any in Pollard and the parallelism in elliptic curves is very coarse and naive. One needs some overall thought about how many curves to try and how to decide upon the coefficients  $A$  and  $B$ , but if one is trying to factor several numbers that are all roughly the same size, then one can choose pretty much the same method to generate the curves to try.

### 14.4.2 Quadratic Sieve:

With the sieve methods we see genuinely interesting possibilities for parallelism.

- With the multiple polynomial sieve we have the naive parallelism of having multiple polynomials to try, and each one can be done indepen-

dently of the others.

- Even within a single polynomial, the sieve array can be broken up into pieces and given to different processors to do different parts.
- In the old days of vector machines, it was important that the stride was constant. This isn't so important now.
- On the flip side is the locality of reference issue. On the one hand, there's a lot of locality for the small primes. Since the stride through the array will be small, there will be lots of hits before we walk out of cache. On the other hand, the small primes don't contribute much to the complete factoring of the residues. And most primes are not small, so most of the time we'll hit once in cache and then have to fetch from another memory block.
- One major positive feature is that we don't really have to be very accurate with the sieve. Rather than use floating point and genuine logs, we can use an approximation (maybe only about 12-16 bits of an integer truncation an approx) to the average value of the log over the interval being sieved. This computation is very robust under failure. We don't in fact have to guarantee that a residue factors completely. We don't have to guarantee that we get all the residues that factor completely. What we really want do is to throw out most of the ones that don't factor completely and keep most of those that will factor completely. If we are good enough at this heuristic, then we can afford to do something more like trial division in a second step, and *that* will be a guaranteed process.

- So we're doing multiple polynomials in parallel, we're doing multiple segments of the sieve array for a given polynomial in parallel, and we're only using short integer arithmetic (i.e., cheap processors) most of the time. And when we do trial division to verify that we have a good line for the matrix, each line can be done entirely independently.
- On the other hand, when we get to the matrix step, we have a probably huge matrix (tens of thousands on a side) on which we want to do a matrix reduction. But it's only 0s and 1s, so once again we can think of using cheap processors as would be available with a massively parallel machine.